PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

PASCAL VERSUS C : A SUBJECTIVE COMPARISON

Prabhaker Mateti

Department of Computer Science
University of Melbourne

ABSTRACT

The two programming languages Pascal and C are subjectively compared. While the two languages have comparable data and control structures, the program structure of C appears superior. However, C has many potentially dangerous features, and requires great caution from its programmers. Other psychological effects that the various structures in these languages have on the process of programming are also conjectured.

"At first sight, the idea of any rules or principles being superimposed on the creative mind seems more likely to hinder than to help, but this is really quite untrue in practice. Disciplined thinking focusses inspiration rather than blinkers it."

- G. L. Glegg,
The Design of Design.

---

1 Introduction

Pascal has become one of the most widely accepted languages for the teaching of programming. It is also one of the most thoroughly studied languages. Several large programs have been written in Pascal and its derivatives. The programming language C has gained much prominence in recent years. The successful Unix operating system and most of its associated software are written in C.

This paper confines itself to a subjective comparison of the two languages, and conjectures about the effect various structures of the languages have on the way one programs. While we do occasionally refer to the various extensions and compilers of the languages, the comparison is between the languages as they are now, and in the context of general programming. The official

documents for this purpose are Jensen and Wirth(1974) and the C-book [Kernighan and Ritchie 1978]. The reader who expects to find verdicts as to which language should be used in what kind of project will be disappointed and will instead find many statements supported only by personal experience and bias; when I felt it necessary to emphasise this, the first person singular is used.

1.1 'Methodology' of Comparison

We do not believe that objective (all-aspect) comparisons of programming languages are possible. Even a basis for such comparison is, often, not clear. (However, see Shaw et al. 1978 .) We can attempt to use such factors as power, efficiency, elegance, clarity, safety, notation, and verbosity of the languages. But elevating these factors from the intuitive to the scientific level by tight definitions renders them useless for the purpose of comparison. For example, all real-life programming languages are as powerful as Turing machines, and hence equally powerful. It is difficult to discuss efficiency of a language without dragging in a compiler and a machine. Furthermore, many of the other notions listed above are based heavily on human psychology, as are the useful insights gained under the banners of structured programming, programming methodology and software engineering. Thus, universal agreement as to the level these notions are supported in a given language will be difficult to reach.

One of the most important factors in choosing a language for a project should be the estimated debugging and maintenance costs. A language can, by being very cautious and redundant, eliminate a lot of trivial errors that occur during the development phase. But because it is cautious, it may increase marginally the cost of producing the first (possibly bugged) version. It is well-known that a programming language affects programming only if the problem is non-trivial and is of substantial size. Also, it seems a language has little effect on the logical errors that remain in a software system after the so-called debugging stage. This is clearly highly correlated with the competence of the programmer(s) involved.

This suggests a method of comparison based on estimating the total cost to design, develop, test, debug and prove a given program in the languages being compared. However, controlling the experiment, and adjusting the results to take care of the well-known effect that the second time it is easier to write (a better version of) the same program (in the same or different language) than to write it from scratch, may prove to be infeasible. Also, very large-scale experiments with a large

piece of software are likely to be so expensive and the results so inconclusive that it is unlikely to be worthwhile. In any case, I do not have the resources to undertake such an experiment.

This comparison is, therefore, necessarily subjective. And this, as can be expected, depends to a large extent on one's own biases, and faith in the recent programming methodology. When the growing evidence supporting this methodology is sufficiently convincing, we can replace the word "faith" by "xxxx".

In the following, we shall

1. compare how "convenient" the languages are to code our favourite solution to a programming problem,

2. play the devil's advocate, and try to list all possible things that can go wrong in a program expressed in a language.

Some of us, including myself, have reservations about the validity of the second technique for comparison, the most persuasive argument being that even though some of the features are potentially dangerous, people rarely use them in those contexts. There is certainly some truth in this, but until we have experimentally collected data convincingly demonstrating this, it is wiser to disbelieve it. Take note of the observed fact of increased difficulty in formally proving the properties of programs that use these potentially hazardous features in a safe way. This is one of the reasons behind the increased redundancy (and restrictions) of the newer languages like Alphard [Wulf et al. 1976], CLU [Liskov et al. 1977], Euclid [Lampson et al. 1977], Mesa [Geschke et al. 1977], and others.

1.2 Hypotheses

It should be clear that neither language is perfect, nor should there be any doubt about the truth of the following:

Axiom [Flon 1975]

> There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.

Since this is a subjective comparison, it is necessary to

identify as many of the underlying assumptions as possible.

1. We believe: (i) That programs should be designed (i.e. conceiving the abstract data structures, and the operations on them, detailing, but not overspecifying, the algorithms for these operations, and grouping all these) in a suitably abstract language, which may not be a formal language. (ii) That the coding (i.e. the translation into a formal programming language) of the abstract program is strongly influenced by the programming language. This paper offers several conjectures about these influences; the word "programming" is used instead of coding, in several places, to emphasise the unfortunate fact that many of us design our programs straight into the programming language.

2. We make a lot of trivial mistakes. Examples : uninitialised or wrongly initialised variables, overlooked typing errors, array indices out of range, variable parameter instead of value, or vice versa, ...

3. The effort spent in physically writing and typing during the development of a large program is negligible compared to the rest of effort.

4. Simple things that could be done mechanically, without spending much thought, should be done by a program.

5. Permissive type checking should be outlawed.

6. It is dangerous to use our knowledge of the internal representation, as chosen by a compiler, of a data type [Geschke et al. 1977].

7. The overall efficiency of a large program depends on small portions of the program [Knuth 1971, Wichmann 1978].

1.3 General Comments

One may wonder: Why compare two languages whose projected images are so different? For example, Sammet's Roster of Programming Languages (1978) lists the application area of Pascal as multi-purpose and that of C as systems implementation.

That Pascal was designed only with two objectives -- viz., a language for teaching programming as a systematic discipline and as a language which can be implemented efficiently -- is quoted often, ignoring four other aims that Wirth(1971) lists. The hidden implication of this attitude is that since Pascal is

suitable for beginners learning to program, it is ipso facto unsuited for adult programming. In fact, an increasing number of complex programs of wide variety from an operating system for the Cray-1 to interpreters on the Intel 8080 are (being) written in Pascal and its dialects.

C is being promoted as a convenient general purpose language. In reviewing the C-book, Plauger(1979) pays his tributes to its authors and claims "C is one of the important contributions of the decade to the practice of computer programming..."

Neither language includes any constructs for concurrent programming. The flexibility of C makes it possible to access nearly all aspects of the machine architecture; low-level programs such as device drivers can thus be written in C. One contention of this paper is that it achieves this flexibility at a great sacrifice of security. Such compromises can be added to Pascal by any implementor, but most have left it relatively pure and unchanged from that described in the revised report [Jensen and Wirth 1974]. Extensions of Pascal to include concurrent programming constructs have resulted in new languages in their own right (Concurrent Pascal [Brinch Hansen 1977], Modula [Wirth 1977a], and Pascal Plus [Welsh and Bustard 1979]).

Thus I believe the domain of application of both languages to be nearly the same.

A great deal of criticism of Pascal has appeared in the open literature ([Conradi 1976], [Habermann 1973], [Lecarme and Desjardins 1975], [Tanenbaum 1978], [Welsh et. al. 1977], [Wirth 1974, 1975, 1977b] and in nearly every news letter of the Pascal User Group [Pascal News]). The little published criticism of C that exists is by people associated with its design and implementation and hence is benevolent. Thus, this paper devotes a greater portion to criticism of C, and repeats some of the criticism of Pascal only when necessary in the comparison.

2. Data Types

One of the greatest assets of both languages is the ability to define new data types. The languages provide a certain number of standard (i.e. predefined) simple types from which other types are constructed. The well-known arrays are composite types whose components are homogeneous. Records of Pascal, structs of C are composite types that (usually) contain heterogeneous components. Other composite types of Pascal that contain homogeneous elements are sets and files. Types are not allowed to be defined recursively, except when they involve a pointer

type. Note that both languages consider a type to be a set of values [Morris 1973].

2.1 Simple Types

Integers, reals, characters, and Booleans are standard types in Pascal. All other types are user defined.

```
type
     zeroto15    = 0..15;
     minus7to7   = -7..7;
     aritherror  = (overflow, underflow, divideby0);
     kindofchar  = (letters, digits, specials);
```

Whereas C has integers, reals, and characters, it does not have Booleans (which is sad), nor does it have a mechanism for defining enumerated types (like the above kindofchar), or subranges (zeroto15). Instead, in some implementations of C, by declaring a variable as short, or char, one obtains smaller sized variables; note the following statement from the C Reference Manual (p182):

> Other quantities may be stored into character variables, but the implementation is machine dependent.

In contrast, the Pascal declarations do not guarantee that smaller units of storage will be used; they simply inform the compiler that it may choose to do so. More importantly, they provide useful documentation; compiling with range checks on, one can have any violations of these caught at run time. In C, this is not possible. The conscious programmer may document the range of some integer variable in a comment, but the compiler cannot help enforce it.

The useful abstraction that Pascal offers in its enumerated types is of considerable value. That this is no more than a mapping of these identifiers into 0..? does not decrease its value. What we, the humans, have to do in other languages, is now done by the compiler, and much more reliably. (It is now rumoured that C will have enumerated types in a future version.)

2.2 Arrays

In Pascal, the index type of arrays is any subrange of scalars (which include enumerated types), whereas in C, arrays always have indices ranging from 0 to the specified positive integer. For example, int a[10] declares an array of ten integers with indices from 0 to 9. Sometimes this leads to

rather unnatural constructs. Consider the following example.

```
line[-1] = '*';       /* any char other than blank,\t, \n */
while  (( n = getline(line, MAXLINE)) > 0) {
    while (line[n] == ' ' || line[n] =='\t' || line[n] =='\n')
        n--;
    line[n+1] = '\0';
    printf("%s\n", &line[0]);
}
```

(In C, = denotes the assignment, == the equality test, and || the McCarthy's OR.)

I find this program clearer, more elegant, and more efficient than the one on p61 of the C-book. However, since arrays cannot have negative indices (as in line[-1]), we are forced to write differently and use a break to exit from the inner loop.

Many people do not appreciate the use of sentinels. Often the argument against them is that you don't have the freedom to so design your data structure. I have not found this to be true in real life situations. This does happen in cooked up classroom situations. It rarely, if ever, is the case that you cannot modify the data structure slightly. The reason for this appears to be a misunderstanding of a fundamental principle of algorithm design :

> Strive to reduce the number of distinct cases whose differences are minor.

The use of sentinels is one such technique. In the above example it guarantees that a non-blank, non-tab, non-new-line character does appear in the array.

The usefulness of negative indices, in these and other situations, should be obvious even to the Pascal-illiterates.

One aspect of Pascal arrays that has come under strong attack is the fact that the array bounds must always be determinable at compile time. This rules out writing generic library routines. There are several suggested extensions to overcome this problem; the signs are that one of these will be incorporated into the language soon.

2.3 Records / Structures

The records and variant records of Pascal are similar to structs and unions of C. However, one important difference must not be forgotten. Pascal does not guarantee any relationships among the addresses of fields. C explicitly guarantees that "within a structure, the objects declared have addresses which increase as their declarations are read left-to-right" (see p196, C-book); otherwise some pointer arithmetic would not be meaningful. Some of the efficiency of pointer arithmetic is provided, in Pascal, by a much safer with statement.

2.4 Pointers

Pointers in Pascal can only point to objects in the heap (i.e., those created dynamically by the standard procedure new), whereas C pointers can point to static objects as well. It is well-known that the latter scheme has the problem of "dangling pointers", and several authors (notably Hoare(1975)) have argued for the abolition of pointers to static objects. The only argument supporting their existence appears to be that they provide an efficient access. It is not known how much this gain in efficiency is in real programs.

On the other hand, unless great caution is exercised, program clarity and correctness are often sacrificed in the process. "A very essential feature of high-level languages is that they permit a conceptual dissection of the store into disjoint parts by declaring distinct variables. The programmer may then rely on the assertion that every assignment affects only that variable which explicitly appears to the left of the assignment operator in his program. He may then focus his attention to the change of that single variable, whereas in machine coding he always has, in principle, to consider the entire store as the state of the computation. The necessary prerequisite for being able to think in terms of safely independent variables is of course the condition that no part of the store may assume more than a single name" [Wirth 1974].

Pascal pointers satisfy the following :

1. Every pointer variable is allowed to point to objects of only one type, or is nil. That is, a pointer is bound to that type; the compiler can still do full type checking.

2. Pointers may only refer to variables that have no explicit

name declared in the program, that is, they point exclusively to anonymous variables allocated by the new procedure when needed during execution.

C pointers, on the other hand, can point to virtually any object -- local, global, dynamically acquired variables, even functions -- and one can do arithmetic on these pointers. The pointers are loosely bound to the type of object they are expected to point; in the pointer arithmetic, each 1 stands for the size of this type. Most C compilers do not check to see that the pointers do indeed point to the right things. Furthermore, the C language definition is such that genuine type confusion occurs. The C-book claims that "its integration of pointers, arrays and address arithmetic is one of the major strengths of the language"; I tend to agree, as their current unsafe setting can be made very secure [Mateti 1979a].

2.5 Type Checking

It is true that one of the basic aims behind the development of strongly typed languages such as Pascal, Euclid, Mesa, Alphard, CLU, etc. is to make it difficult to write bad programs. In realising this goal, all programs become slightly more difficult to write. But this increase in difficulty is of a mechanical kind, as we now expect the programmer to provide a lot of redundant information.

Type checking is strongly enforced in Pascal, and this is as it should be. Errors caused by incompatible types are often difficult to pinpoint [Geschke et al. 1977]. Strong type checking does increase the time to produce the first version of a syntactically correct program, but this is worthwhile. It is true that Pascal has not satisfactorily defined when two types (with different names) are equivalent [Welsh et al. 1977] but the problems are easily avoided by appropriate declarations. Any required type conversion occurs only through predefined functions for the purpose, or through user-defined variant records. (The latter are unsafe; see Section 8.)

In sharp contrast, all kinds of type conversions are either blessed in C, or ignored by its compilers. For example, our Interdata 8/32 C compiler detected only one error in the program of Figure 1. In fact, it is rare that you see a C program that does not convert the types of its variables, the most common conversion being that between characters and integers. More recently, however, C designers have provided a special program called lint that does type checking. A few points should be

```
main()
{
                /* See Section 2.5                          */
  int           /* integer                                  */
    i,
    xx,
    a[10],
    f(),        /* f is a function returning integer        */
    (*pf)();    /* pointer to a function returning int      */

  printf(" exponent part of 123.456e7 is %d \n",
        expo(123.456e7));

  i = a;        /* i now points to a[0]                     */
  a[1] = f;     /* a[1] points to the function              */
  2[a] = f();   /* 2[a] is equivalent to a[2]               */
  a[3] = f(0);  /* f called with 1 argument                 */

  pf = &xx;     /* pf now points to xx                      */
  i  = (*pf)(); /* now call the "function" pointed to by pf
                                                           */

  a = i;        /* This is the only illegal statement       */
                /* in this program caught by C compiler     */
                /* because a is not a left-value.           */

}

f(a,b)          /* f in fact has 2 formal parameters        */
char    a, b;
{
   if  (a)  return (b);
}

expo(r)                 /* see Section 9.2                  */
float   r;
{
        static struct s {
                char c[4];      /* uses 4 bytes             */
                float f;
        } c4f;
        static char *p = &(c4f.c[3])+1;
                /* points to first byte of f                */

        c4f.f = r;
        return(*p);
}
```

Figure 1

remembered in this context:

1. Type compatibility is not described in the C Reference Manual. Presumably this is similar to that of Pascal, and Algol 68. It is not clear exactly what it is that lint is checking.

2. The need for type checking is much greater during program development than afterwards. In fact, a good argument can be made that the primary goal of a compiler is this kind of error checking and code generation its secondary goal; the function of type checking should be an integral part of a compiler. To separate it from the compiler into a special program whose use is optional is a mistake, unless it is a temporary step.

3. Type checking is not something that you can add on as an afterthought. It must be an integral part of the design of the language.

It is fair to say that type conversion is difficult in Pascal but frequent need for this is a sign of bad program design. The occasional real need is then performed by explicit conversions.

2.6 Control of Storage Allocation

It is possible to specify in Pascal that certain variables be packed thereby saving storage. The semantics of such variables is the same as if they were regular variables. There are standard procedures to unpack. It should be noted that specifying packing simply gives permission to the compiler to pack; however, the compiler may decide otherwise.

C does not have a corresponding facility. But C structures can have "fields" consisting of a specified number of bits. These fields are packed into machine words automatically. It is also possible to suggest that a variable be allocated a register.

3. Statements and Expressions

C is an expression language, a la Algol 68, in a limited way; only assignments, function calls, special conditional expressions have values. Thus , for example, a function that does return a value can be called like an ordinary procedure in C, which would be illegal in Pascal, as Pascal is strictly a statement language. Below, we take a more detailed look at these

aspects.

3.1 Boolean Expressions

C does not have genuine Boolean expressions. Where one normally expects to find these (e.g., in if and while statements), an ordinary expression is used instead, and a non-zero value is considered "true", a zero being "false". Relations yield 1 if true, 0 if false. The operators & and | are bitwise AND and OR operators, && and || are similar to McCarthy's "logical" AND and OR operators : x && y is equivalent to the conditional expression if x ≠ 0 then y ≠ 0 else 0 fi and x || y is equivalent to if x = 0 then y ≠ 0 else 1 fi. For example,

4 & 6	is	4	4 && 6	is	1
4 & 8	is	0	4 && 8	is	1
4 | 6	is	6	4 || x	is	1
4 | 8	is	12	0 || x	is	(x ≠ 0)
			0 && x	is	0

where x is any expression, including the undefined one. The operators &, | are commutative, but &&, || are not. The left-to-right evaluation of the "logical" operators && and || of C does save, occasionally, a few micro-seconds. The traditional AND and OR operators have a nice property that they are commutative, in conformity with their use in mathematics. As a consequence, any reasoning we do using them is more readily understandable. One specific outcome of the use of the unorthodox operators is that the many cases where both the operands are indeed evaluated have to be discovered by involved inferences. A better solution is to have logical operators of the traditional kind, reserving the McCarthy's operators for use when really needed. To my mind, even when these McCarthy's operators are really required , to spell them out as in

```
if  B1  then
        if  B2  then
                .....
```

is much more readily understandable. I suspect this to be the main reason behind the warning " Tests which require a mixture of &&, ||, !, or parentheses should generally be avoided." of the C-book(p61).

3.2 Assignments

The symbol denoting the assignment operator is = in C; it is rumored that this was a conscious choice as it means one less character to type. Pascal uses the conventional left arrow, written as :=. C allows assignments to simple (i.e., non-struct, non-array) variables only, at the moment; structure-to-structure and array-to-array assignments are among its promised extensions. The assignment statement has the same value as that assigned to the left hand side variable; thus, we can write conveniently,

```
i = j = 0;
```

Pascal allows assignments to whole arrays as well as records. However, the assignment is not an expression, and the above has to be expanded as:

```
i := 0;
j := 0;
```

3.3 Operator Precedence

C has over thirty "operators" (including (), [], ., the dereferencing operator *), and fifteen precedence levels, compared to Pascal's six arithmetic operators, four relational operators and four precedence levels. Because of the many levels, and also because some of them are inappropriately assigned, one learns to survive either by constantly referring to the C manual and eventually getting them by rote, or by over-parenthesising; for example,

```
x & 07 == 0      is equivalent to    x & (07 == 0)
*++argv[0]       is equivalent to    *++(argv[0])
```

The basic problem is that the operators like &, or && take any integers as operands, and a missing pair of parentheses will result in a meaningful but unexpected expression.

It is neccessary to parenthesise in Pascal also, but here the reason is different : there are too few levels, as arithmetic operators and boolean operators got merged in their priority. For example,

```
flag and a < b
```

would result in type incompatibility, which should be written as

```
                    flag  and  (a < b)
or as,              (a < b) and flag
```

using commutativity of Pascal and.

3.4 The Semicolon

Pascal uses the semicolon as a statement separator, whereas C uses it as a statement terminator. It is well-known that statement separators are the cause of many syntax errors in beginner's programs [Nutt 1978]. But it rarely is a problem for the experienced; most of us have learned to use it as a terminator (with a null statement following).

4. Control Structures

> Control structure is merely one simple issue, compared to questions of abstract data structure.
>
> - D. E. Knuth (1974)

For the last ten years or so, the literature concentrated on control structures, and we have learned enough to cope with their abstraction. Some significant rules of thumb have emerged; e.g. use procedures extensively, keep them short, avoid gotos, never jump out of a procedure. As a result, control structures play a rather local role; they are important, but their effect can be localised to these short procedures. Data structure abstraction is not well-understood, in sharp contrast to their design and choice. Many of the remaining errors in large software systems, after an initial period of development, can be attributed to "interface problems" which can be roughly described as inconsistent assumptions about data structures in different places. With this perspective, we move on to the control structures of the two languages.

4.1 Looping

In C, loops are constructed using while, do-while, and for. To exit prematurely from a loop, a break is used; to terminate the current iteration but continue from the next, continue is used. Similar loop structures in Pascal are, respectively,

while, repeat-until, and for; premature termination can be accomplished only by gotos. But there is a world of difference between the for statements of the two languages.

The C for statement duplicates what can be done by other structures with equal clarity;

```
for (expr1; expr2; expr3) statement
```

is an abbreviation of

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

Note that the three general expressions can be arbitrarily complex. A missing expr2 is equivalent to specifying the constant 1 as expr2.

The Pascal for statement is an abstraction of an often occurring structure ;

```
for i := first to last do statement
```

loops exactly last - first + 1 times, if first <= last, or not at all. The control variable i starts with a value of first, takes successive values up to last. The values last, first, and variable i are all of a scalar type. A downward for is constructed by using downto instead of to. There have been suggestions in the literature [Hoare 1972] that the variable i should be a read-only variable local to the body of the loop; Pascal compromisingly insists [Addyman et al. 1979] that the variable be local to the procedure/function/program block in which the for loop occurs.

4.2 Selection

The if statements of the two languages are very similar except that C uses general expressions, as in while-statements, instead of Boolean expressions, and the word then is omitted.

```
case expression of
    cl1  : S1;
    cl2  : S2;
    ...
    cli  : Si;
    ...
end;
T
```

The above case statement of Pascal transfers control to one (say Si) of several statements whose constant case label cli equals the value of the scalar expression. When the execution of Si terminates, control is transferred to T. If the expression value does not match any case label, the effect of case statement is undefined in standard Pascal. A default label cannot be given either; several implementors have felt the need for this, and it is now allowed on most implementations. However, it should be emphasised that in most well-written programs the expression value belongs to an enumerated type which is exhaustively listed by the case labels cli.

The switch statement of C is primarily used to create a similar effect. However, control passes from Si to the next Si+1, unless this flow is explicitly broken by a break :

```
switch  (exp)  {
    case  L1   : S1
    case  L2   : S2
                 break;
          ...

    case  Li   : Si
    case  Li+1 : Si+1

          ...

    default    : Sn
}
T
```

The "usual arithmetic conversion" is performed on exp, if necessary, to yield an integer value. The labels Li must be manifest integer expressions. If exp matches no Li, it matches the default and if the optional default label is absent, then none of the statements in the switch is executed. Whereas a default label is wanted in Pascal, it is needed in C as it cannot be hoped that the case labels Li will exhaust the values that exp can take.

Note also that C needs a break because the only way to group cases is by falling through cases. For example, to combine more

than one case, say 2 and 4, you write

```
case 2   :
case 4   :
         ----
         ----
         break;
```

and in such situations Pascal does not need a break, as labels can be grouped. The C-book wisely cautions (p56), "...falling through cases is a mixed blessing....Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly."

4.3 The Power of Control Structures

Loops with break/continue belong to the class DREC1, in the genealogy of control structures [Ledgard and Marcotty 1975]. Theorems by Kosaraju(1974) show that a DREC1 structure cannot be simulated by D-structures (D for Dijkstra), which are formed by any number of ifs, whiles, and concatenation using original variables, actions and predicates. In fact, some DRECi structures (which contain BLISS-like [Wulf et al. 1971] multi-level exit(i), exiting i enveloping loops, and cycle(i), continuing the next iteration of the i-th enveloping loop) are more powerful than any DRECi-1 structures.

With this background we make the following observations :

1. This does not mean that a given problem, for which we have a solution with break/continues, cannot be solved using D-structures only but with a different choice of data structures. In fact, most breaks used in the programs of C-book can be so avoided; some of them occur only because the array index cannot be negative.

2. Why stop at break and continue, which are equivalent to exit(1) and cycle(1) ? Certainly, for i > 1, exit(i) and cycle(i) add flexibilty and power. The primary function of control structures is to provide clarity by operational abstraction. Loops containing exits, and cycles are more difficult to understand. It is surprising how rarely one really needs exit(1) or higher exits. The need for control structures at higher levels than D-structures is still unproven.

3. But, if one feels a break is needed in a certain situation,

why not use a goto? Knuth(1974) argues that such use of goto is not "unstructured", while a lot of others (like Ledgard and Marcotty(1975)) would rather introduce a boolean variable, or expand the range of values of an already existing variable, to eliminate the break.

Both languages have the goto statement. In Pascal, the labels need to be declared, and are always unsigned integers. C allows arbitrary identifiers as labels, which are not declared.

5. Program Structure

Pascal and C both have a simpler program structure than Algol 60. Pascal achieves simplicity by identifying blocks with routines (procedures/ functions) and C does it by not allowing nested routines. In spite of this, C program structure, particularly the scope of variables, is more comprehensive than that of Pascal. Successors to Pascal, such as Concurrent Pascal, Modula, Pascal-Plus, have successfully blended into Pascal the notion of Simula-classes, which structures programs far more effectively.

5.1 Procedures and Functions

In Pascal, these are two distinct entities. A function returns a scalar, real, or pointer value, but has no side-effects when well-written. When a procedure is called, we expect the environment to change; when a function appears in an expression, we can evaluate it without at the same time worrying about side-effects. This is how it should be in a statement-oriented language.

C functions, on the other hand, may or may not return values. In the latter case, they are equivalent to Pascal procedures. But, C goes one step further, and permits a variable number of parameters and the use of value-returning functions as procedures. Certainly, it is more natural for some routines to have a variable-number of parameters (e.g., Pascal's read and write). But this should be the exception allowed only upon explicit request.

Another surprise in C is all the parameters are passed by value only. To achieve Pascal's var parameter, the address is passed as a value parameter, and the function changes the content of the cell pointed. Thus C depends too heavily on pointers, providing a classic case of type confusion as in

```
char *s ;
```

(Is s a pointer to a character, or a pointer to an array of characters?)

5.2 Block Structure and Scope

C does not allow nested functions but the body of any compound statement is a block and can contain declarations (of struct, typedef, and variables). This feature, however, is rarely used in practice, except in the outermost block of a routine, or when register variables are needed. Such block structure can be simulated in Pascal by calls to nested routines, but this incurs the overhead of a call.

The names of functions in C are always global (unless declared static) and available to routines in other source files. Variables and new type names can be declared in between routines, or before the very first one, and are visible to routines below them in that file. To access variables declared in other files, explicit extern declarations are required. Variables can be declared, within a routine, to belong to the static storage class (similar to own variables in other languages); such variables retain their values between successive calls of that function and are visible only within that routine. These features and the ease of separate compilation make it possible to structure C programs with as much clarity (but not security) as can be achieved with the module concept. In contrast, such structuring cannot be done elegantly in Pascal.

6. Language Support

It is clear to anyone involved in the production of software that often the support given to a language plays a more major role than the language itself. Supporting tools include source language debugging packages, execution profilers, cross-reference generators, macro (pre)processors, pretty printers, and a host of other library programs. To be sure, none of these is part of a language, but most users cannot distinguish them as being separate entities because of their careful integration into host languages.

C is a good example of this process. It uses a standard preprocessor for handling constant definitions and file inclusions. Many of these tools for C are written in C, and hence available just as widely as the language itself. In

contrast, Pascal tools and separate compilation facilities [Kieburtz et al. 1978] are only now being developed by interested users. Some of these are written in non-standard Pascal and often integrate poorly with operating systems.

6.1 Preprocessors

Pascal programmers often get annoyed by the lack of some simple conveniences. Examples:

1. Expressions involving symbols defined at compile time cannot be used on the right hand side of a constant definition :

   ```
   const  n = 10;  n1 = 11;
   ```

   If we change n to, say, 20, then we should also change manually n1 to 21. The following is simpler and more informative :

   ```
   const  n = 10;  n1 = n + 1;
   ```

   but this is illegal.

2. Body substitutions for calls to (very short) functions and procedures cannot be specified. The grouping of short sequences of tests and other operations into functions and procedures is thereby discouraged.

Both situations are quite common in programming, and to argue that they can be done easily by hand, and that execution profiles often prove that body substitutions do not yield space/time gains is simply unrealistic.

C handles the above situations, as well as inclusion of text from other files, excellently through its standard macro preprocessor. Such a processor is easy to write for Pascal too, but as there is no standard syntax for it, too many different preprocessors are bound to mushroom [Comer 1979, Mateti 1979b].

7. Efficiency

> "Don't diddle code to make it faster -- find a better algorithm."
>
> - Kernighan and Plauger (1974)

We can distinguish between two kinds of efficiency improvements: of the algorithm, of the coding. The efficiency that complexity theorists discuss often deals with the asymptotic behaviour of the execution time of algorithms. When input data are of sufficiently large size n, an O(n) algorithm would in fact be faster than an O(n**2) algorithm. This may, however, not always be the case on small amount of input data. If you have only a five element array to sort, bubble sort may run faster on your machine than O(n log n) quick sort.

Also, the following appears to be the case, unless the algorithm in question is a well-studied one :

> The lower the level of the language, the more afraid you are to use a more complex but significantly more efficient algorithm.

However, the practising programmer often appears overly concerned with improving efficiency only at the statement-level of coding. This penny-wise saving of micro-seconds has an apparently incurable side-effect that the resulting programs are harder to understand and often incorrect. Not uncommonly, more significant global improvements are not realized because of the unmastered complexity introduced at this statement-level. This is the direct result of incomplete analysis of the program written.

The benefits of a theoretical complexity analysis are very often substantial. But leaving this aside, one can further distinguish two kinds of efficiency improvements at the coding level :

1. measurable improvements
2. demonstrable improvements

For example, let us take a millisecond as the unit of measurement. Then, these are not always the same -- 1. implies 2., but not vice versa; for, you may be able to demonstrate that program A is faster than B by executing them a thousand times and comparing the total execution times, even though A is not

measurably faster than B.

We should not ignore another observed phenomenon that programs spend most of their time in very small portions of the code. If this is true of the program in question, try to improve the efficiency of only these small segments of the code.

Correctness-preserving efficiency improvements, of whatever kind, should certainly be followed provided the required effort is not too great and the resulting code is equally easy to understand, maintain, enhance and modify. When this proviso is not satisfied a careful analysis of the benefits of efficiency improvements is necessary. For example, is it worthwhile to (demonstrably) improve a program that runs only a few times a day? Is not a millisecond too small a unit for distinguishing the two kinds of improvements for cost benefits?

By providing such things as register variables, and decrement and increment operations, C gives the impression of being an efficient language. We have, as yet no solid evidence that this is so, or if so, by what factor, in the domain of systems programming. For example, the absence of negative indices for arrays and the lack of sets induces more computation than is actually necessary. While it is true that i++ can be compiled straightforwardly into demonstrably faster code than i := i + 1, it is not clear if such things make programs measurably faster. On the other hand, there is the real danger of a slight slip turning such a statement into a major disaster (see Sections 9 and 10.1).

> "It is very easy to exaggerate the need for efficiency and require a performance competitive with optimal hand coding."
>
> - B. A. Wichmann(1978)

8. Portability

Perhaps the too restrictive nature of Pascal and the ease with which its compilers can be modified are the two factors that prompt many of its implementors to 'extend' the language and make it unportable. (Is giving rise to a host of suggested extensions a charactersitic of a superior language?) But programs in standard Pascal enjoy a considerable degree of portability (apart from problems caused in any language by the underlying character

codes, ASCII or EBCDIC, or whatever).

This cannot be said of C. Even though most of the existing compilers are built by a rather close-knit group at Bell Labs and MIT, there are enough differences. One reason for this may be that the semantics of the language is often confused with what code the compilers produce in its Reference Manual.

Certainly, C programs have been and can be ported [Johnson and Ritchie 1978]. But this does not mean that they are portable as the word is generally understood. There is no clearly defined subset of it that would guarantee portability. A few example problems that the C Reference Manual cautions about are :

1. A pointer can be assigned any integer value, or a pointer value of another type. This can cause address exceptions when moved to another machine.

2. Integers can be assigned to chars and vice-versa.

To these we can add the problems caused by assumptions made in C programs about the addresses of variables (that they are a fixed distance apart ...). The unions of C and variant records of Pascal can both cause portability problems when misused.

9. Insecurities

"For the purpose of this discussion, an insecurity is a feature that cannot be implemented without either (1) a risk that violations of the language rules will go undetected, or (2) run-time checking that is comparable in cost to the operation being performed" [Welsh et al. 1977]. It may sound paradoxical but few or no insecurities need not always be a good thing. For, we observe that assembly languages have no insecurities whatsoever, according to the above definition, for the simple reason that it does not attempt to provide any security. It is only when a language purports to provide security, either explicitly or implicitly, and then fails that we should be upset by it. Thus, we modify (1) to read 'a risk that violations of the language rules and intentions will go undetected'. It is unlikely that a useful language without any insecurities can ever be designed. We can attempt to reduce their number, and explicitly identify them so that we are not lulled into believing that programs written in the language are safe.

9.1 Unsafe Features

An unsafe feature is an insecurity that generally causes havoc and is frequently the cause of evasive bugs.

The list of unsafe features of C is rather long : pointers to static as well as dynamic variables, address arithmetic, passing addresses as value parameters, treating an object pointed to as an array, all belong to this list. But what is more important is that they constitute the most heavily used features. Some of these exist in the language purely for the sake of statement-level efficiency. The use of pointers in accessing array elements is not only efficient, but has a certain elegance of its own. However, its setting is extremely unsafe, and provides much fuel to the "pointers considered harmful" debate (e.g. [Hoare 1975]). It is possible to control the use of pointers without any loss in efficiency [Mateti 1979a]. As they are now, they can be greatly misused, worse, an accidental slip can turn it into a very frustrating and harmful gremlin.

Not only is the list of unsafe Pascal features short -- variant records without tag fields, functions and procedures as parameters, dangling pointers to dynamic variables -- their relative frequency of occurrence is far lower.

9.2 Dirty Tricks

A dirty trick is an exploitation of an insecurity. The adjective "dirty" is used only to remind that such tricks often spring up as a nasty surprise to any one but their originators. Contrary to popular belief, dirty tricks can serve clean and legitimate purposes. This happens when the language is put to use in a way its designer has not foreseen or wished to forbid but could not. More often, however, they provide short-cuts. Two such examples follow.

1. Suppose we wish to access the exponent part e of the representation of a positive real number x. On the Interdata 8/32, this happens to be in bits 1 to 7. Thus, the function expo of Figure 1 would do the job in C.

2. Suppose we wish to produce the 32-bit concatenation of four 8-bit quantities, or vice versa. On some machines, characters are represented as 8-bit bytes and integers as 32-bit words. Thus, declare the 8-bit quantities as characters, and

```
var dummy :
    record case  boolean  of
        true  : (bits32 : integer);
        false : (bits8  :
          packed array [1..4] of char);
        end;
  ...
  ...
with  dummy  do begin
    bits8 [1] := first  8-bit quantity;
    bits8 [2] := second 8-bit quantity;
    bits8 [3] := third  8-bit quantity;
    bits8 [4] := fourth 8-bit quantity;
    end;
```

then dummy.bits32 is the required concatenation. Code similar to this appears in some Pascal compilers. Pascal chose deliberately to provide this flexibility at the expense of security [Wirth 1975].

10. Psychological Effects

W e a r e a l l p s y c h o l o g i s t s.

- from a book on psychology

It is with some trepidation that I write on these effects, for I am a computer scientist. However, to shy away from this "non-subject" would be to ignore the recognised importance [Weinberg 1971] of the effects caused by our mental images of the languages and by our human limitations. If you are sceptical of what is said here, you are justified. But, I urge you to test these hypotheses out and see how true/false they are.

10.1 Error Proofing

> That the ratio of all "meaningful" constructs to all syntactically legal constructs in any programming language is almost zero

is a well-known fact. This is not because the said programming language is defined in a context-free grammar rather than in a more precise one such as vW grammar [Tanenbaum 1978]. (It is

possible, by technical trickery, to define a "programming language" where this ratio is unity; such a language would, however, have extremely limited "expressive power".) Let us recall the assumptions of Section 1.3. In addition, the following appear to be true, but are not well-tested:

> The number of errors in programs is proportional to the amount of detail that the writer had to handle in his program.
>
> The cost of debugging is a rapidly increasing function of the number of errors(bugs), which includes the extremely trivial ones.

It is therefore important to decrease the possibilities for (unintentional) misuse. Thus it is desirable to inform the compiler of our intentions.

> How can we expect a language to aid in avoiding mistakes, if it is even incapable of assisting in their detection.
>
> - N. Wirth (1974)

10.2 Understandability and Compactness

> Programming is the art of writing essays in crystal-clear prose and making them executable.
>
> - P. Brinch Hansen (1977)

It can justifiably be argued that the code is not a complete source of information about a program and that a programmer understands a program by successively refining guesses about how the program operates [Brooks 1978]. However, we confine ourselves here to the understanding gained through reading the code only.

Programs in expression languages are (to me) more difficult to understand than those in statement languages. In the latter, only the statements are active in modifying the values of variables. It is for this reason that we often discourage functions with side-effects. In understanding expression language programs we have to handle more details at different levels all at the same time. We need to remember not only what the expression value is so far, but also what variables have which new values. It is also true that expression language

programs are more compact. Thus, we remark that

> Readability is inversely proportional to compactness.

This is not to say anything verbose is readable. The word compactness, as it is used here, needs explanation. Electronic circuits can be made more compact by using integration. But this does not make them less complex than their discrete component counterparts. Compactness achieved in expression languages is of this kind. Unlike in mathematics, where compact notation hides detail irrelevant to a given level of discussion, expression language programs while being compact still contain all the gory details. The algorithm does not become simpler, nor is there any reduction in the number of abstract operations except that in the code generation some redundant load/store machine instructions may be avoided.

In C, a programmer can certainly choose not to be compact but

> the natural tendency of most programmers to write the "best possible" code in a given language works against writing readily understandable code.

Do give some thought to the qualification in the following quote.

> C is easy to write and (when well-written) easy to read.
>
> McIlroy et al. (1978)

> However, although we are all psychologists at heart, not all of us are scientists.
>
> - from the same book on psychology

11. Conclusion

The images that Pascal and C evoke are vivid. The strength of C emanates from its identification of several practices used in assembly programming that lead to very well-written, modular,

and efficient programs. In addition, C provides a modern syntax for them adding the conventional wisdom of high-level languages, notably automatic allocation of storage for variables and recursion. Its fundamental flaw is that it failed to curb the misuse of the very same features. While "misuse" is relative to one's programming "morals", the failure to provide enough redundancy to catch the accidental slip is unrealistic and can be expensive.

Pascal, on the other hand, gives the impression that it may have been designed by first synthesising all that has been put forward in its time about "good and wholesome" programming, and eliminating features that cannot be implemented efficiently enough. Its promotion and exposition may have been, from a psychological point of view, offensive : restrictions are often resented, and rarely understood. It is true of nearly every human endeavour that it takes far greater courage, training, education and understanding to be disciplined, and computer programming is no exception.

Optimism has not, apparently, worked in the past programming projects. "Its [software] products have typically contained other than what was expected (usually less, rather than more), been delivered much later than scheduled, cost more than anticipated, been poorly documented, and been poorly designed" [Bersoff et al. 1979]. One should learn this lesson, and be extremely careful at every step. Languages with convenient features whose erroneous use cannot be detected by its compilers should be avoided.

That excellent (as well as extremely ugly) programs can be written in either language is clear. However, I am concerned that it is all too easy to write incomprehensible programs in C. Even more offending are the "features" such as unbridled pointers, variable number of parameters in function calls, absence of type checking and lack of Boolean variables... ; these are a lot more troublesome than they are worth.

Finally, let me conclude by quoting Welsh et. al.(1977):

> "Pascal is at the present time the best language in the public domain for purposes of systems programming and software implementation.
>
> The discovery that the advantages of a high-level language could be combined in such a simple and elegant manner as in Pascal was a revelation that deserves the title of breakthrough. Because of the very success of Pascal, which greatly exceeded the expectations of its

author, the standards by which we judge such languages have also risen. It is grossly unfair to judge an engineering project by standards which have been proved attainable only by the success of the project itself, but in the interests of progress, such criticism must be made."

Acknowledgements

Many discussions with Paul Dunn, Robert Elz, Ken McDonnel, and Peter Poole prompted me to think about this topic and write this paper. However, they may not share my views as expressed here. I am grateful to the many authors who have influenced me and whose quotations I have so heavily used to make it clear that this paper is little more than a collage of their ideas.

12. References

[Addyman et al. 1979]
A. M. Addyman, et al., "A Draft Description of Pascal," Software - Practice and Experience, Vol. 9, No. 5, 381 - 424.

[Bersoff et al. 1979]
Edward H. Bersoff, Vilas D. Henderson and Stan G. Siegel, "Software Configuration Management : A Tutorial," IEEE Computer Magazine, Vol. 12, No. 1, 6 - 13.

[Brinch Hansen 1977]
Per Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall.

[Brooks 1978]
Ruven Brooks, "Using a Behavioral Theory of Program Comprehension in Software Engineering," Proceedings of the Third International Conference on Software Engineering, IEEE, 196 - 201.

[Comer 1979]
Douglas Comer, "MAP : A Pascal Macro Preprocessor for Large Program Development," Software-Practice and Experience, Vol. 9, 203 - 209.

[Conradi 1976]
R. Conradi, "Further Critical Comments on the Programming Language Pascal, Particularly as a System Programming Language," SIGPLAN Notices, Vol. 11, 8 - 25.

[Flon 1975]
Lawrence Flon, "On Research into Structured Programming," ACM SIGPLAN Notices, Vol.10. No. 10, 16-17.

[Geschke et al. 1977]
Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite, "Early Experience with Mesa," Communications of the ACM, Vol. 20, No. 8, 540-553.

[Habermann 1973]
A. N. Habermann, "Critical Comments on the Programming Language Pascal," Acta Informatica, Vol 3, 47 - 58.

[Hoare 1972]
C. A. R. Hoare, "A Note on the for Statement," BIT, Vol. 12, 334-341.

[Hoare 1975]
C. A. R. Hoare, "Data Reliability," ACM SIGPLAN Notices, Vol. 10, No. 6, 528-533.

[Jensen and Wirth 1974]
Kathleen Jensen and Niklaus Wirth, Pascal : User Manual and Report, 2nd ed., 4th printing, Springer-Verlag, pp 167.

[Johnson and Ritchie 1978]
S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the UNIX System," The Bell System Technical Journal, Vol. 57, 2021 - 2048.

[Kernighan and Plauger 1974]
Brian Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill.

[Kernighan and Ritchie 1978]
Brian Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall Software Series, pp viii + 228.

[Kieburtz et al. 1978]
R. B. Kieburtz, W. Barbash and C. R. Hill, "A Type-checking Program Linkage System for Pascal," Proceedings of the Third International Conference on Software Engineering, IEEE, 23 - 28.

[Knuth 1971]
Donald E. Knuth, "An Empirical Study of FORTRAN Programs," Software-Practice and Experience , Vol. 1, 105-133.

[Knuth 1974]
Donald E. Knuth, "Structured Programming with goto Statements," Computing Surveys, Vol. 6, No. 4, 261 - 301.

[Kosaraju 1974]
Rao Kosaraju, "Analysis of Structured Programs," J. Computer and System Sciences, Vol. 9, No. 3, 232 - 255.

[Lampson et al. 1977]
B. W. Lampson, J. J. Horning, R. L. London, J.G. Mitchell, and G. L. Popek, "Report on the Programming Language Euclid," ACM SIGPLAN Notices, Vol. 12, No. 2, pp ii + 79.

[Lecarme and Desjardins 1975]
O. Lecarme and P. Desjardins, "More Comments on the Programming Language Pascal," Acta Informatica, Vol. 4, 231 - 243.

[Ledgard and Marcotty 1975]
Henry F. Ledgard and Michael Marcotty, "A Genealogy of Control Structures," Communications of the ACM, Vol. 18, No. 11, 629 - 639.

[Liskov et al. 1977]
Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, "Abstraction Mechanisms in CLU," Communications of ACM, Vol. 20, No. 8, 564-576.

[Mateti 1979a]
Prabhaker Mateti, "Enumerated Types and Efficient Access of Array Elements," in preparation.

[Mateti 1979b]
Prabhaker Mateti, "Specifications of a Macro Preprocessor for Pascal : A CS340 Project," University of Melbourne.

[McIlroy et al. 1978]
M. D. McIlroy, E. N. Pinson and B. A. Tague, "Foreword (to the special issue)", The Bell System Technical Journal, Vol. 57, No. 6, 1899 - 1904.

[Morris 1973]
J. H. Morris, "Types are not Sets," Conference Record ACM Symposium on Principles of Programming Languages, Boston, Mass., 120 - 124.

[Nutt 1978]
Gary J. Nutt, "A Comparison of Pascal and FORTRAN as Introductory Programming Languages," ACM SIGPLAN Notices, Vol. 13, No. 2, 57-62.

[Pascal News 197x]
Pascal News, News letters of the Pascal Users Group, Andy Mickel (ed.), University of Minnesota.

[Plauger 1979]
P. J. Plauger, A Review of Kernighan and Ritchie 1978 , Computing Reviews, Vol. ?, 2 - 4.

[Sammet 1978]
Jean E. Sammet, "Roster of Programming Languages for 1976-1977," ACM SIGPLAN Notices, Vol. 13, No. 11, 56-85.

[Shaw et al. 1978]
Mary Shaw, Guy T. Almes, Joseph M. Newcomer, Brian K. Reid and Wm. A. Wulf, "A Comparison of Programming Languages for Software Engineering," Report CMU-CS-78-119, Carnegie-Mellon University.

[Tanenbaum 1978]
A. S. Tanenbaum, "A Comparison of Pascal and Algol 68," The Computer Journal, Vol. 21, 316 - 323.

[Weinberg 1971]
Gerald Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold.

[Welsh and Bustard 1979]
J. Welsh and D. W. Bustard, "Pascal-Plus - Another Language for Modular Multiprogramming," Australian Computer Science Communications, Vol. 1, No. 1, 49 - 62.

[Welsh et al. 1977]
J. Welsh, W. J. Sneeringer and C. A. R. Hoare, "Ambiguities and Insecurities in Pascal," Software - Practice and Experience, Vol. 7, 685 - 696.

[Wichmann 1978]
B. A. Wichmann, "Some Performance Aspects of System Implementation Languages," Constructing Quality Software, P. G. Hibbard/S. A. Schuman (eds.), IFIP, North-Holland, 46 - 62.

[Wirth 1971]
Niklaus Wirth, "The Design of a Pascal Compiler," Software -- Practice and Experience, Vol. 1, 309-333.

[Wirth 1974]
Niklaus Wirth, "On the Design of Programming Languages," (in) Information Processing 1974, J. L. Rosenfeld (ed.), North-Holland, 386-393.

[Wirth 1975]
Niklaus Wirth, "An Assessment of the Programming Language Pascal," Proceedings of 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol. 10, No. 6, 23-30.

[Wirth 1977a]
Niklaus Wirth, "Modula : A Language for Modular Multiprogramming," Software - Practice and Experience, Vol. 7, 3-35.

[Wirth 1977b]
Niklaus Wirth, "Programming Languages : What to Demand and How to Assess Them," (in the book) Software Engineering, edited by R. H. Perrott, Academic Press, 155 - 173.

[Wulf et al. 1971]
W. A. Wulf, D. E. Russell, and A. N. Habermann, "BLISS : A Language for Systems Programming," Communications of the ACM, Vol. 14, No. 12.

[Wulf et al. 1976]
W. A. Wulf, R. L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," IEEE Transactions on Software Engineering, Vol. 2, 253-265.